Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics

William Killian[a], Renato Miceli[*b,c], EunJung Park[a], Marco Alvarez Vega[a], John Cavazos[a]

[a]*University of Delaware, USA*
[b]*Irish Centre for High-End Computing (ICHEC), Ireland*
[c]*Université de Rennes 1, France*

Abstract

Vectorization support in hardware continues to expand and grow as we still continue on superscalar architectures. Unfortunately, compilers are not always able to generate optimal code for the hardware; detecting and generating vectorized code is extremely complex. Programmers can use a number of tools to aid in development and tuning, but most of these tools require expert or domain-specific knowledge to use. In this work we aim to provide techniques for determining the best way to optimize certain codes, with an end goal of guiding the compiler into generating optimized code without requiring expert knowledge from the developer. Initially, we study how to combine vectorization reports with iterative compilation and code generation and summarize our insights and patterns on how the compiler vectorizes code. Our utilities for iterative compilation and code generation can be further used by non-experts in the generation and analysis of programs. Finally, we leverage the obtained knowledge to design a Support Vector Machine classifier to predict the speedup of a program given a sequence of optimization. We show that our classifier is able to predict the speedup of 56% of the inputs within 15% overprediction and 50% underprediction, with 82% of these accurate within 15% both ways.

1. Introduction

Vectorization support in hardware continues to expand and grow as we still continue on superscalar architectures. Unfortunately, compilers are not always able to generate optimal code for the hardware; detecting and generating vectorized code is extremely complex. The Intel Compiler has its own internal heuristics used to determine whether or not code should be vectorized, but they do not provide direct access for the developer to see why the compiler chooses one optimization over another. Vectorization reports can help relay information from the compiler to the programmer. Some compilers also provide the capability for the programmer to aid the compiler with vectorization with the use of compiler-specific pragmas.

The number of tools a programmer can use to aid in development continues to grow, but most of these tools require expert or domain-specific knowledge to use (e.g. vectorization reports). We aim to provide an iterative compilation method of determining the best way to optimize certain codes. Intel offers a Vectorization Optimization Guide [16] for their architecture and development tools, but it cannot directly carry over to other platforms. Stock et al. [28] propose using machine learning techniques to improve automatic vectorization. Machine learning techniques have also been applied to entire programs and optimization search spaces to improve execution [7][24]. This existing work is inefficient to our goal because they still require expert knowledge to set up or optimizations cannot be applied at a granular enough level (per loop).

We aimed to provide a look at how we can combine vectorization reports with iterative compilation and code generation to show insights and patterns on how the Intel Compiler vectorizes code. We explored various vectorization optimizations to apply to a given code. Our goal is to provide insights which carry across various benchmarks and codes to help developers guide the compiler into generating good code without necessarily having expert knowledge.

Our experiments consisted of iterative compilation of a benchmark suite designed for the evaluation of vectorizing compilers [20] and generated vectorization reports for each code. Categorization and mapping of different microkernels into different classes were performed to help classify them by how the compiler was able to improve performance. The same experiment was

* Corresponding author. E-mail address: renato.miceli@ichec.ie

then done on a more complex set of kernels designed for polyhedral compilers as a way to see how the work scales to larger programs.

To aid with iterative compilation, we created two utilities. One is a source-to-source compiler which translates a simplified directive language to a specific directive language supported by a given compiler (e.g. Intel Compiler). The other utility performs iterative compilation among a set of optimizations to apply more than once in a given program. The creation of these utilities can help non-experts in the generation and analysis of programs.

Finally, we leveraged the knowledge we obtained with the tuning work to specify a machine learning classifier to predict the speedup of a benchmark given a sequence of optimizations. We could then use our SVM-based predictor to choose the optimization sequence to apply that yields the best speedup. Experiments show that our classifier predicted the speedup of 56% of our benchmarks within 15% overprediction and 50% underprediction, with 82% of these accurate within 15% both ways.

The results of this work can help programmers tailor their applications to take the most out of their vectorizable codes. The iterative compilation utilities can be especially useful when exploring tuning options during the code optimization phase. The SVM-based predictor can be used as part of a general-purpose auto-tuning strategy that does not require human interaction.

In this work we make the following contributions:

1. **`VALT`** – a directive compiler used to simplify code generation across different compiler backends;
2. **`autovec`** – an iterative compiler utility to generate different versions of the same code;
3. Insights and categorization of benchmarks based on vectorization reports, speedup, and accuracy of different versions; and
4. An SVM-based speedup predictor capable of predicting the speedup of simple loop nests within 15% accuracy 45% of the time.

The rest of the paper is outlined as follows: Section 2 goes over the command-line and directive-based compiler optimizations which can be applied to a program. Section 3 introduces the kernels used in the evaluation of our work. Section 4 describes our **`VALT`** and **`autovec`** tools used for version generation. Our vectorization analysis is illustrated in Section 5. The SVM predictor is introduced and evaluated in Section 6. We go over related work and future work in Sections 7 and 8, respectively. The paper ends with the Conclusions and Acknowledgements, highlighting the main contributions of this work.

2. Intel Compiler Optimizations

The Intel C++ Compiler version 14.0.2 offers many optimization options for the programmer. Some of these are restricted to command-line flags, while others must be inserted as directives into the original code. This section addresses the relevant optimization flags and directives used in this work.

2.1 Command-Line Flags

All configurable programmer-passed optimizations happen through command-line flags. When performing code optimization, optimization-level flags (**`-O2`**, **`-O3`**) tend to be the most commonly used. In addition to specifying a set of optimizations to apply, the programmer is also able to specify the target architecture and modify the default code generation rules. Listed below are relevant command-line flags used in this work.

- **`-O3`** – apply all **`-O1`** and **`-O2`** optimization sets in addition to a new set. A full list of these optimizations are viewable from the Intel C++ Compiler reference guide [17].
- **`-xHOST`** – specify the architecture that we are optimizing for is the same one we are compiling on. This is equivalent to specifying the native architecture on your system. For our experiments, **`-xAVX2`** would be applicable to the Haswell (HSW) microarchitecture and **`-xSSE4.2`** would be applicable to the Nehalem (NHM) microarchitecture.
- **`-vec`** / **`-no-vec`** – specify whether or not any generated code should be vectorized. **`-vec`** permits any amount of generated code to be vectorized while **`-no-vec`** prevents any code from being vectorized.

We used the command-line flags to drive native code generation for the platform we were executing on.

2.2 Source Code Directives

In addition to command-line flags, the Intel Compiler is also capable of processing programmer-placed directives (in C/C++ written as **`#pragma <opt>`**). Many of these are specific to the Intel Compiler; however, some of them are gaining support in other compilers such as GCC [12].

Here, we will look at a subset of directives offered in the Intel Compiler, specifically those that aid with vectorized code transformations and generation.

- **`#pragma vector always`** – compiler will ignore the speedup factor when considering vectorization. If the compiler believes the code will execute slower with vectorization, it will still vectorize the loop.
- **`#pragma ivdep`** – compiler will ignore all *unproven* interloop dependences. Note that all *proven* dependences will not be vectorized.
- **`#pragma simd [vectorlength(n)]`** – compiler will ignore all dependences and reductions. Everything related to vectorization is left for the programmer to manage. An optional argument for **`simd`** is specifying a vector

length. This value is passed to the compiler to state how many safe iterations can be done at once.

Such as in previous works [9] we used the source code directives to drive the optimization selection and modification to show that we can guide the vectorization heuristics to improve performance.

3. Kernels

To evaluate the Intel Compiler's built-in vectorization heuristics, a set of benchmarks were used to determine performance improvement. Two sets of benchmarks were used for this work. The first set, *Test Suite for Vectorizing Compilers* [20] is an extension and modification of a test suite for vectorizing Fortran compilers in the late 1980's [4]. The second set of kernels are the *Polybench* kernels stemming from Pouchet's work with Polyhedral Compilers [27].

3.1 Test Suite for Vectorizing Compilers

The Test Suite for Vectorizing Compilers contains over 150 different loop nests which iterate over different access patterns, computations, and memory access types (e.g. single value, (un)aliased pointers). The test suite was designed to evaluate how well compilers were able to recognize patterns which could be vectorized and apply them. For this work, we were using this test suite to evaluate and see which built-in heuristics in the Intel compiler may not enable the best performance while maintaining numerical correctness. By issuing varying directives, we were able to relax Intel Compiler's built-in heuristics and instead see how applying optimizations, perhaps unsafely by the compiler's view, adjusts the performance of the loop nest. For sake of code re-use and to simplify generation and execution, each loop nest was placed in its own file.

3.2 Polybench

Polybench/C 3.2 contains 30 different static control-flow micro-benchmarks deriving from several scientific domains (e.g. linear algebra, machine learning, image processing). As with the Test Suite for Vectorizing Compilers, we have used these kernels to explore the potential increase in performance. No modifications were made to the original kernels; only directives were programmatically inserted.

4. Version Generation

We developed two utilities in order to help with version generation for iterative compilation. **`autovec`** is a source-to-source compiler which translates a simplified directive language to a specific directive language supported by a given compiler (e.g. Intel Compiler). **`VALT`** performs iterative compilation among a set of optimizations to apply more than once in a given program. These utilities can help nonexperts in the generation and analysis of programs, and have been used here to understand the inner workings of the compiler's vectorization strategies.

4.1 ***`autovec`***

For version generation of each kernel, we used scripts with placeholder directives to drive which optimization was to be used on a per-loop basis. We defined our own directive language, **`autovec`** (autovectorizer), to aid with this task.

`autovec` directive	**Intel-specific pragma**
`permute`	*generate each version*
`vl(x)`	`simd vectorlength(x)`
`always`	`vector always`
`ivdep`	`Ivdep`
`none`	<nothing>

Table 1: **`autovec`** directive support and translation

Table 1 shows the possible directives we can parse and generate with **`autovec`**. When **`autovec`** is given a permute argument before a loop, the tool will permute through all loop optimizations and generate a different version. As multiple loops are issued with this command, the number of possible versions of a single kernel grows exponentially.

Benchmarks	Versions
3	6
125	36
21	216
2	1,296

Type	Count
Total	11,646
Correct	10,026
Incorrect	1,612
Compile Error	8

Table 2: Code version size and analysis for TSVC

For TSVC, the number of versions explored can be observed in Table 2. Version generation can generate unsafe code as well. Table 2 also shows versions of code which are *(a)* safe and adhere to the original implementation, *(b)* unsafe and cause a variance in result, and *(c)* invalid and could not even compile.

For Polybench we had to reduce our search space intelligently as one kernel had 10 different loops within a single loop nest. Since some vectorization directives cannot be applied in a nested manner, we were able to reduce the search space by three orders of magnitude.

4.2 ***VALT***

An extension to **`autovec`** was also created with additional optimizations also permitted. **`VALT`** (vectorization and loop transformation) enables a developer to quickly specify which vectorization and loop transformation directives to apply to a given loop. Backends exist for both Intel compiler (Intel-specific pragmas) and CAPS compiler [5] (**`hmppcg`** directive support).

`VALT` directive	**Intel-specific pragma**
`vector(default)`	*<no code emmited>*
`vector(none)`	`Novector`
`vector(always)`	`vector always`
`vector(ignore)`	`Ivdep`
`vector(aligned)`	`vector aligned`
`vector(temp)`	`vector temporal`
`vector(nontemp)`	`vector nontemporal`
`vectorsize(x)`	`simd vectorlength(x)`
`loop(unroll(x))`	`unroll(x)`
`loop(jam(x))`	`unroll and jam(x)`
`loop(nofusion)`	`Nofusion`
`Loop(dist)`	`distribute point`

Table 3: VALT directive language translation for Intel-specific pragmas

Table 3 shows how **`VALT`** directives directly translate to Intel-specific pragmas. For this work we limited our use of **`VALT`** to only be used for vectorization directives.

5. Vectorization Analysis

Intel Compiler v14.0.2 was used for all experiments throughout this research. Table 4 shows the machine configuration used for both the analysis and speedup predictor.

	Analysis	**Predictor**
Processor	i7-4960HQ	i7-950
Clock Rate	2.6GHz (3.8GHz)	3.06GHz
L3 Cache	6MB	8MB
On-Chip Memory	128MB	–
Shared Memory	16GB DDR3-1600	24GB DDR3-1333

Table 4: Machine Configurations

All time measurements were performed in cycles so any issue related to dynamic frequency scaling affecting performance was mitigated. Speedups were measured as compared to the "default" optimization configuration: no added directives and compiled with **`-O3 -xHOST -vec`**. Intel vectorization reports **`(-vec-report6`**) were also obtained to aid in classification and to see why the compiler chose not to vectorize some loop nests.

```
(17): loop was not vectorized: existence of vector dependence
(19): vector dependence: assumed ANTI dependence a(19) and a(18)
(18): vector dependence: assumed FLOW dependence a(18) and a(19)
(18): vector dependence: assumed FLOW dependence a(18) and a(19)
(19): vector dependence: assumed ANTI dependence a(19) and a(18)
(15): loop was not vectorized: not inner loop
(13): loop was not vectorized: not inner loop
```

Figure 1: A sample vectorization report

Figure 1 shows a sample vectorization report where the Intel Compiler did not vectorize the loop because of assumed vector dependences and being unable to vectorize non-inner loops. The original benchmark's vectorization report was compared to each

new version's vectorization report. With this additional information, we were able to identify patterns and trends based on

1. How the optimizations applied affect performance
2. Which loop nests the Intel Compiler did not want to vectorize
3. Why the Intel Compiler did not want to vectorize
4. How the optimizations applied affected further loop optimization (e.g. loop unrolling, additional vectorization)

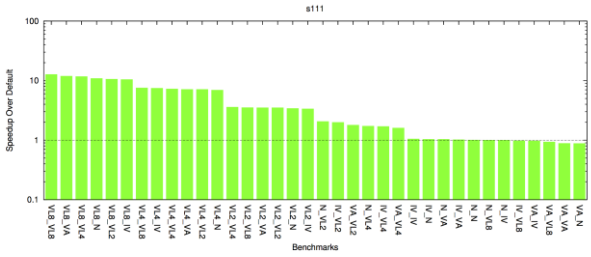
5.1 TSVC

There were 27 loop nests where the optimizations with the best speedup were unsafe – that is the compiler heuristics correctly prevented unsafe code from being generated. That said, 124 loop nests had optimizations with the best speedup being safe, with one in particular executing over 50× faster than the default.

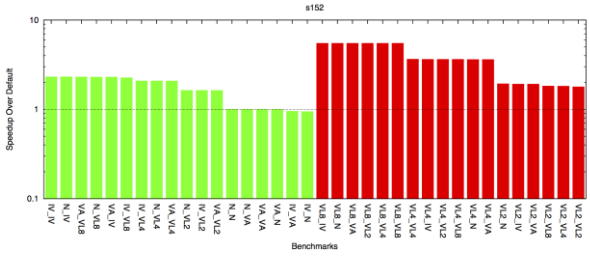
Speedup	Count	Total
> 1.00×	16	151
> 1.01×	30	135
> 1.05×	22	105
> 1.10×	23	83
> 1.50×	12	60
> 2.00×	24	48
> 4.00×	16	24
> 8.00×	8	8

Table 5: Best speedup by iterative compilation

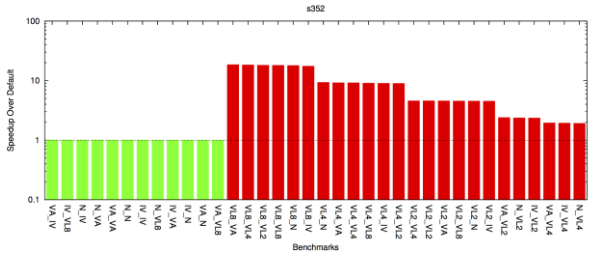
Table 5 shows that 105 loop nests achieved a speedup greater than 5%. For speedups less than 8× the primary reason for improved speedup was due to reduced cache misses and SIMD instruction execution. For the case of one benchmark (**`s257`**) where the speedup was 57×, improved data access, SIMD instruction execution, and reduced TLB/page lookups were observed.



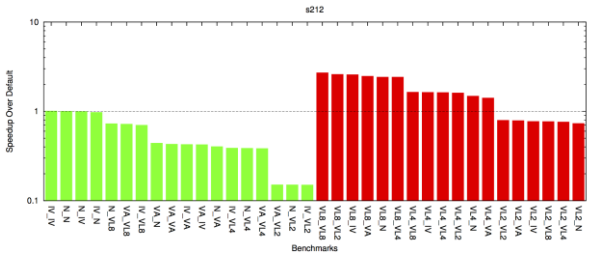
(a) Good speedup observed; no invalid code generation



(b) Good speedup observed with invalid code faster



(c) No speedup observed; invalid code generation faster



(d) No speedup observed; invalid code generation

Figure 2: TSVC loop nest optimizations and speedup comparison (valid code generation is in green; invalid is in red)

Figure 2 shows sample performance analysis of four different TSVC loop nests and demonstrate varying level of correctness. Figure 2a shows a loop nest where good speedup was observed with no invalid code generation. This is the ideal case. Figure Figure 2b shows a loop nest with good speedup observed, but the fastest speedup occurred with invalid code. Likewise, invalid code generation can occur with no speedup observed with valid code generation as shown in Figure 2c. Sometimes, no variation with valid code generation produced a speedup. In this situation illustrated by Figure 2d, any optimization either does nothing or produces invalid code.

For loops which saw low (or no) speedup improvements, there were a few varying reasons why.

1. Loops were not vectorizable by the nature of the computation. This caused unsafe code generation (and therefore excluded).
2. Loops were already highly optimized by the compiler. Sometimes what the Intel Compiler did for code generation was already the best. This could be observed by comparing **`-vec`** and **`-no-vec`** generated code to the best-optimized code.
3. Improved code generation for SIMD instructions caused a bandwidth and data access issue which increased the

number of stalls from cache invalidation. This was verified with profiling and comparing the generated optimized assembly code to the default.

Furthermore, we were able to classify TSVC loop nests into a few different categories.

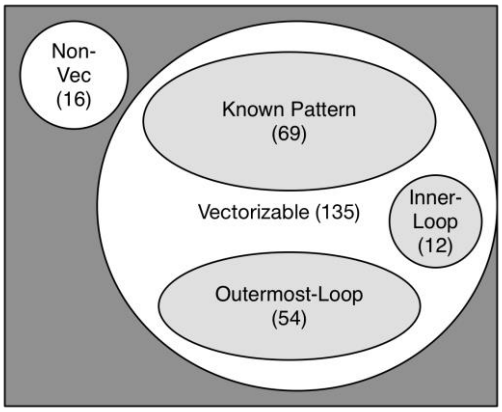


Figure 3: Categorization of TSVC loop nests

Figure 3 gives an overview of the category composition while the description for the categories is found here:

1. *Non-vectorizable* – Loop nests which were not vectorizable. There were 16 benchmarks (11%) that are not vectorizable as indicated by the optimized version vectorization report and minimal/non-existing speedup observed.
2. *Known vectorization pattern* – Loop nests which could be vectorized by the compiler with minimal additional speedup observed after optimizing. A total of 69 benchmarks (46%) fell into this category. This suggests that overall, the Intel Compiler is able to vectorize code well with its built-in heuristics although they are not always optimal.
3. *Inner-loop vectorizable* – Loop nests which were not initially vectorized well but were better optimized with a **`#pragma simd`** directive placed in an inner loop. 12 of the loop nests (8%) were observed to be inner-loop vectorizable with a speedup of at least 2×.
4. *Outermost-loop vectorizable* – Loop nests which were not initially vectorized well but were better optimized with a **`#pragma simd`** directive placed in the outermost loop of the loop nest. 54 of the loop nests (35%) were observed to be outermost-loop vectorizable with a speedup of at least 2×.

5.2 Polybench

Polybench kernels are more complex than the TSVC loop nests. First, they can have more than one loop nest. Second, they can operate on multiple data elements at one time, potentially causing cache contention with different data structures at the same time. Polybench kernels can be grouped into a few different categories for the vectorization analysis:

1. Known kernel – The loop nest and access pattern match a known kernel which the Intel Compiler knows how to optimize well. **`2mm`**, **`3mm`**, and **`gemm`** are all known kernels of the Intel compiler. All three of these benchmarks had extremely high speedups when the **`-no-vec`** baseline was compared against the **`-vec`** vectorized code. Although they had high speedups when compared to **`-no-vec`**, almost no speedup was observed when **`-vec`** was compared to the best optimization sequence.
2. Known loop nest – The loop nest structure is identified by the compiler and the optimization is performed. **`trmm`**, **`gemver`**, **`gesummv`**, **`symm`**, and **`mvt`** are all kernels where some or all of their loop nests are known by the compiler. Here a portion of the code is highly optimized for the access pattern. Unfortunately, sometimes the compiler's heuristics prevent some optimizations from being applied (see **`symm`** and **`mvt`**).
3. Known access pattern – The access pattern structure is identified and loop reordering is performed by the compiler to improve vectorization. **`atax`** and **`trisolv`** are examples of kernels where the compiler is aware of the access pattern and can optimize accordingly. **`trisolv`** has a triangular access pattern while **`atax`** performs a multiplication and transpose – both are access patterns which the Intel Compiler can optimize.
4. Unknown – The compiler's heuristics could not classify it into any of the three prior mentioned categories. The remaining kernels in general fall into this category. Here the loop nest structure cannot easily be optimized due to iteration dependence. If forced to vectorize, the generated code would most likely be incorrect. These are the kernels where the loop analysis heuristics of the compiler are used to determine potential optimizations.

For every kernel which was not known, the best optimization sequence yielded unsafe optimizations. For most kernels, there was not a large improvement in speedup with the optimization search space.

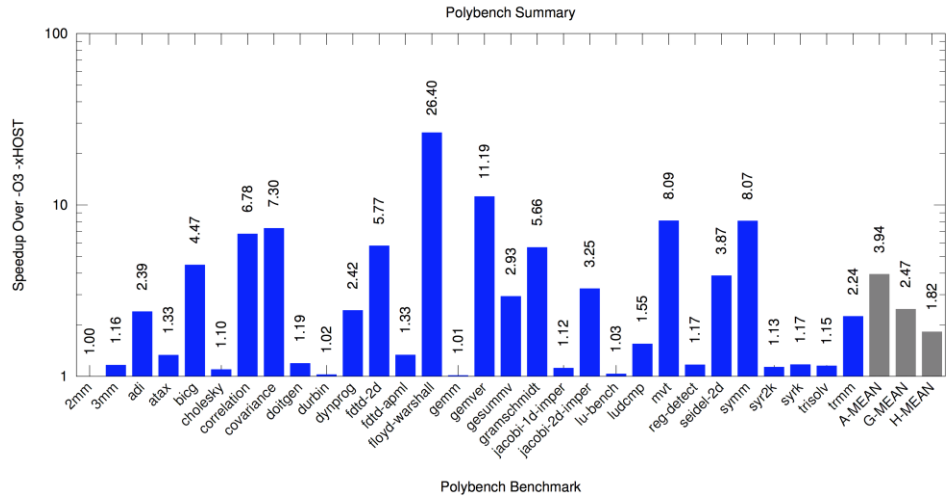


Figure 4: Best speedup by iterative compilation (incl. invalid code generation) for Polybench (mean results are in gray)

Speedup	Type	Right	Miss	Wrong	Kernels
< 1.05×	Known	5	2	3	`durbin, gemm, 2mm, 3mm, cholesky, lu-bench, jacobi-1d`
< 1.10×	Loop Nest	4	3	2	`trmm, gemver, gesummv, mvt, symm, syr2k, syrk`
< 3.00×	Access	4	1	2	`trmm, gemver, gesummv, mvt, symm, syr2k, syrk`
> 3.00×	Unknown	6	2	4	`adi, bicg, correlation, covariance, dynprog, fdtd-2d, floyd-warshall, grammschmidt`

Table 6: Polybench kernel classification and speedup ranges

Table 6 shows the correlation between the classification of the Polybench kernels to a speedup range. By following the guidelines established the different classifications, speedup thresholds were used to determine how accurate each classification would be based on type. On average we were correct 50% of the time when we consider all kernels. When we ignored those that were wrong and instead focused on how they were classified and missed, the classification rate was 70%.

6. SVM-Based Speedup Predictor

Given the speedup information based on different optimizations for a collection of loop nests and kernels, an end goal would be to automate the prediction of the speedup of a benchmark given some input data and the optimization sequence targeted to a particular code.

6.1 Performance Counters

Performance counters have been used by Cavazos et al. [7] and Park et al. [24] for optimization selection on a larger scale, but never at a per-loop basis. We collected 45 different performance counters categorized into one of the following information areas:

- L1, L2, and L3 cache information
- TLB information
- Cache line access
- Branch instructions
- Floating-point instructions and SIMD
- Load-store instructions
- Cycle/Interrupt/Stalls/Instruction information

We used PAPI [2] for the automated collection of performance counters during execution. We modified a single, common header file across the benchmarks to include the performance counter selection so no benchmark codes needed changes.

6.2 Experiment Configuration

We used the 151 TSVC loop nests as training data for our SVM-based speedup predictor model. For training with a support vector machine (SVM) classifier, we need to specify our feature vector. Our feature vector consists of the 45 different performance counters normalized to the number of instructions executed, the speedup of the vectorized code over non-vectorized

code, the optimization bit vector, and the speedup of the optimized code over vectorized code.

The feature vector is described in greater detail including composition in Figure 5. The optimization bit vector format is described in Table 7. Each additional bit set indicates a stricter level of optimization.

Bit Configuration	Optimization
00000	No Optimization
10000	`#pragma vector always`
11000	`#pragma ivdep`
11100	`#pragma simd vectorlength(2)`
11110	`#pragma simd vectorlength(4)`
11111	`#pragma simd vectorlength(8)`

Table 7: Optimization bit vector configuration

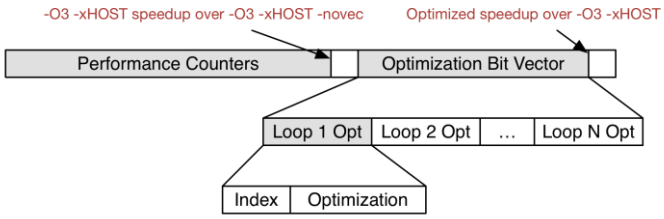


Figure 5: SVM feature vector description and format

Multiple loops for a given code would increase the number of optimization bit vectors. Because of potential ambiguity, we elected to have the optimizations listed in source code line order. For the case of TSVC loop nests, no loop nest exists for size greater than 4. To simplify training and classification, any loop nests of size less than 4 were padded with 0 indicating no further optimization to be applied.

Table 4 presents the machine configuration we used to collect all training data for prediction.

6.3 Prediction Results

The predictor model we used targets single loop nests, so using the Polybench programs with this model was not applicable. Instead, we favored the TSVC loop nests to see how well our support vector machine model would be able to predict speedup. We tested our trained models with leave-one-out cross validation (LOOCV). For a given loop nest found in the TSVC set, we constructed a model based on the other 150 loop nests and compared the model's prediction to the actual speedup observed for the model. The same procedure was done for all other loop nests.

To evaluate our model, we defined a metric to compare our predicted speedup to the best actual speedup observed. This metric is defined as:

$$E = \frac{S_{best} - S_{pred}}{S_{pred}}$$

When E is less than 0, then our predictor overestimated speedup. Likewise, any value greater than 0 indicates our predictor underestimated the speedup.

Percentile	Type	Count
0.15	Under	45
0.50	Under	15
1.00	Under	12
2.00	Under	16
> 2.00	Under	27
0.15	Over	24
> 0.15	Over	12

Table 8: Speedup Predictor Accuracy

Table 8 shows how our predictor performs with the count of each benchmark in each area. 84 of the 151 loop nests were accurate within 15% overprediction and 50% underprediction. 5 of the overpredictions were on loop nests which were non-vectorizable.

The SVM-based predictor cannot accurately predict speedup for over 44% of the loop nests. For 46% of the loop nests, the

speedup predictor is accurate within 15%. Analysis of the types of loop nests and their predicted speedup did not show a correlation between the types of benchmarks which were under-predicted vs. over-predicted.

7. Related Work

The closest work related to this research is Stock's article on *Using machine learning to improve automatic vectorization* [28]. The major limiting contribution of this work is the scope of the class of loops which the model is able to optimize (tensor and stencils). Their training model is derived from information within the inner-most loop and vectorization information. A takeaway from their work was the importance of *(a)* a single feature alone does not correlate to good performance and *(b)* a weighted rank model always outperformed SVM. Additional work has been divided into three main categories: *Vectorization*, *Compiler Optimization*, and *Machine Learning*.

7.1 Vectorization

Callahan [4] created an initial set of micro-benchmarks to evaluate the vectorization capability of compilers. The benchmarks were written in Fortran and the case study was performed at the infancy of SIMD in consumer, general-purpose hardware. Maleki et al. [20] extended the original benchmark suite to support C as well as the addition of several more micro-benchmarks. Maleki then manually modified some codes that did not vectorize which could ultimately be vectorized. Our work builds on Maleki's contribution by automatically relaxing vectorization heuristics to properly vectorize certain benchmarks.

Henretty [13] was able to improve vectorization of stencil applications with a compiler focused on stencil codes, but does not extend to other types of kernels. Nuzman [23] improved performance of kernels with outer-loop vectorization on CBE and PowerPC architectures. Our work extends this by targeting Intel microarchitectures automatically with the use of the **`#pragma simd`** directive.

Holewinski [15], Evans [10], and Barik [3] used trace information to determine vectorization potential and automate the selection of vector instructions. Hohenauer [14] proposed a framework to enable retargetable compilers to emit more appropriate SIMD instructions. McFarlin et al. [21] automatically vectorized FFT kernels. Both of these works focused on modifying or constructing a compiler to perform the optimizations. McFarlin's work is limited in scope to FFT kernels while Hohenauer's work cannot be easily extended to new architectures or compilers.

7.2 Compiler Optimization

Kong et al. used Polyhedral transformations to help drive improved vectorization [18]. This method can help improve performance, but it does not help determine where or why existing compilers cannot better optimize certain code. Pouchet et al. [26] used iterative and model-driven optimizations to drive auto-parallelization. The same can be done for auto-vectorization.

7.3 Machine Learning

Park et al. [24][25], Dubach et al. [8], and Cavazos et al. [6][7] focused on using machine learning methods to construct prediction models based on performance counters. Park extended the existing work by using graph-based program characterization. Agakov [1] used machine learning to reduce and eliminate branches of optimizations being applied. Work has also been done in the area of finding which features to use for machine learning models for optimized compilation [19].

8. Future Work

We plan to continue development of **`VALT`** to support multiple compiler backends and extend **`autovec`** to other types of directive-based optimizations. PGI Compilers have their own directive language; we could construct a backend for **`VALT`** to emit the PGI directives for cross-compiler analysis on the same architecture. **`autovec`** could also switch from an iterative code generator to an auto-tuner capable of some intelligent selection of optimizations to explore, thus we could consider its inclusion in general-purpose auto-tuning frameworks [11][22].

We intend to switch to a graph-based speedup predictor as the SVM-based predictor did not perform as well as we hoped. Using a graph-based speedup predictor would also allow to extend the classes of micro-benchmarks to go beyond simple loop nests (those provided in TSVC) and extend to non-perfectly-nested loops (such as Polybench).

Due to increased support of vectorization directives such as **`#pragma simd`** and **`#pragma ivdep`** we will be able to extend this existing work directly to compilers such as GCC 4.9. In addition to considering other compilers, we could explore architectures with wider vectorization sizes such as the Intel Xeon Phi (Knight's Corner) and the upcoming Knight's Landing microarchitecture where AVX-512 is supported in GCC and Intel Compiler.

Conclusions

In this paper we provided techniques, both manual and automatic, for determining the best way to optimize programs regarding their vectorization capabilities. Initially, we studied how to combine vectorization reports with iterative compilation and code generation and summarized our insights and patterns on how the compiler vectorizes code. The utilities we developed for iterative compilation and code generation can be further used by non-experts in the generation and analysis of programs. Finally, we leveraged the obtained knowledge to design a Support Vector Machine classifier to predict the speedup of a program given a sequence of optimization. We showed that our classifier is able to predict the speedup of 56% of the inputs within 15% overprediction and 50% underprediction, with 82% of these accurate within 15% both ways.

The results of this work can help programmers tailor their applications to take the most out of their vectorizable codes. The iterative compilation utilities can be especially useful when exploring tuning options during the code optimization phase. The SVM-based predictor can be used as part of a general-purpose auto-tuning strategy that does not require human interaction. Overall, the contributions introduced in this paper can help programmers guide the compiler into generating optimized code without requiring expert knowledge on the compiler inner workings or the underlying architecture.

Acknowledgements

This work was supported by the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no. RI-283493. The authors would like to thank the FP7 AutoTune project (grant agreement no. 288038) for their collaboration in this work.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 11 pp.–, March 2006.

[2] I. C. L. at University of Tennessee. Performance application programming interface. Online, Dec 2014.

[3] R. Barik, J. Zhao, and V. Sarkar. Automatic vector instruction selection for dynamic compilation. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 573–574, New York, NY, USA, 2010. ACM.

[4] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 98–105, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[5] CAPS Entreprise. Caps compilers. Online, May 2014.

[6] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. P. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 24–34, New York, NY, USA, 2006. ACM.

[7] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, pages 185–197, March 2007.

[8] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimization evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pages 131–142, New York, NY, USA, 2007. ACM.

[9] B. Eagan, G. Civario, and R. Miceli. Investigating performance benefits from openacc kernel directives. In M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 616–625. IOS Press, 2014.

[10] G. C. Evans, S. Abraham, B. Kuhn, and D. A. Padua. Vector seeker: A tool for finding vector potential. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 41–48, New York, NY, USA, 2014. ACM.

[11] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, A. D. Malony, Z. Chamski, D. Novillo, and D. Del Vento. Collective mind: Towards practical and collaborative auto-tuning. *Scientific Programming*, 22(3), Sept. 2014.

[12] GCC Team. Gcc 4.9 release series changes, new features, and fixes. Online, May 2014.

[13] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[14] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. A simd optimization framework for retargetable compilers. *ACM Trans. Archit. Code Optim.*, 6(1):2:1–2:27, Apr. 2009.

[15] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI '12, pages 371–382, New York, NY, USA, 2012. ACM.

[16] Intel Corporation. A guide to auto-vectorization with intel c++ compilers. Online, April 2012.

[17] Intel Corporation. User and reference guide for the intel c++ compiler 14.0. Online, September 2013.

[18] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA, 2013. ACM.

[19] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning-based optimising compilation. *ACM Trans. Archit. Code Optim.*, 11(1):14:1–14:32, Feb. 2014.

[20] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, Oct 2011.

[21] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. Automatic simd vectorization of fast fourier transforms for the larrabee and avx instruction sets. In *Proceedings of the International Conference on Supercomputing*, ICS'11, pages 265–274, New York, NY, USA, 2011. ACM.

[22] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In P. Manninen and P. Öster, editors, *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2013.

[23] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[24] E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 196–206, New York, NY, USA, 2012. ACM.

[25] E. Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 119–129, April 2011.

[26] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010.

[27] L. N. Pouchet. Polybench/c: the polyhedral benchmark suite. Online, March 2012.

[28] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim.*, 8(4):50:1–50:23, Jan. 2012.